Linear time construction of suffix trees

We will now describe Esko Ukkonen's algorithm for constructing suffix trees in linear time (Ukkonen 1995). Note that the suffix tree structure was developed earlier by Weiner and subsequent work by McCreight and Ukkonen simplified the construction approach.

The following description can be quite confusing. I suggest you also use online resources to better understand what's going on. The following link http://stackoverflow.com/questions/9452701/ukkonens-suffix-tree-algorithm-in-plain-english provides a nice description and also a number of other pointers.

The basic idea of the approach is that we will iteratively build the tree on prefixes of the string. For example, for string BANANAS, we will build the tree for string B, then for BA, BAN, and so on. We'll denote by T_i the suffix tree built on the ith prefix of the string S (T_1 corresponds to B in our example, T_3 is BAN, etc.) At each stage in the process we will add one character to all the suffixes stored in the earlier trees, as well as add the final suffix for the just added character. Note that we will ignore the $ character for now, and allow suffixes to end along a path in the tree.

Clearly this algorithm appears to be quite expensive. We process n different prefixes, and at each prefix i we have to update i different suffixes, which leads to an $O(n^2)$ algorithm. If we take into account the time needed to 'find' each of the suffixes we may end up with an $O(n^3)$ solution, far from what we are trying to achieve.

A couple observations can help us speed up this algorithm dramatically. For the following, assume we are processing suffix α of tree T_i and try to extend it with character S[i+1].

1. **Once a leaf, always a leaf.** Once the algorithm has created a leaf, that leaf will never disappear. Furthermore, the string labeling the edge leading to this leaf will always end at the end of the current string (position i in the original string for tree T_i) As a corollary, at each stage in the algorithm we do not need to update the leaves that are already created. Instead, once we create a leaf we can simply label the end of the edge with a special 'end of string' character (remember, the edge is simply a pair of numbers corresponding to the coordinates of the edge in the original string). Once the tree is constructed we will spend O(n) time to replace these numbers with n – the end of the whole string. Also note that there are O(n) leaves in the tree and we only spend computation when creating each leaf, thus the total time spent on leaves in our algorithm is overall O(n).

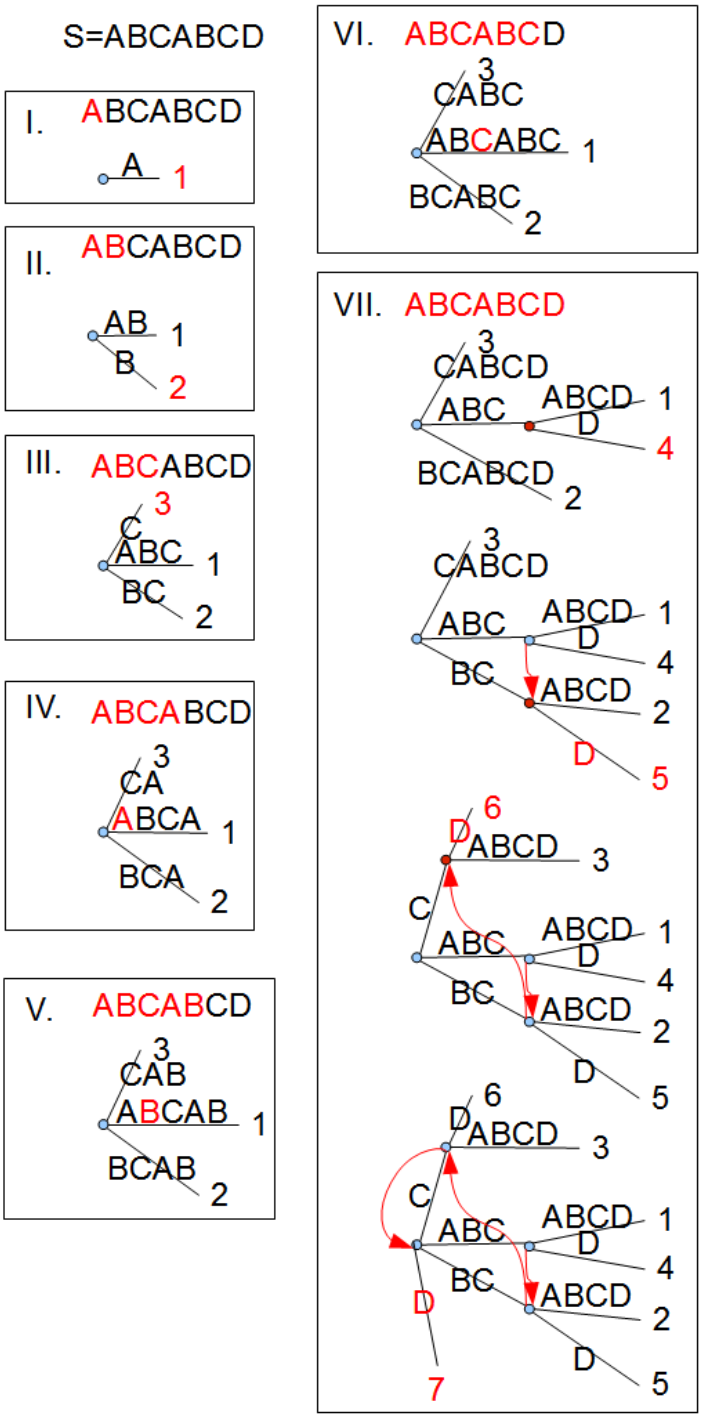
2. **If string αS[i+1] already exists in the tree, all subsequent suffixes also exist.** This observation is fairly easy to prove. If αS[i+1] existed in tree T_i, then clearly all other suffixes shorter than it would also be in T_i by the definition of this tree. As a result, when the our algorithm reaches this situation, we can simply stop and proceed with the next prefix of S, i.e., tree T_{i+1} is complete and we can proceed to T_{i+2}. As a corollary, we can also bound the number of times we encounter this situation (a suffix of T_{i+1} is already in T_i) to O(n) as the situation only occurs once per iteration.

3. **If a node is created by the algorithm, the node to which the suffix link will point either exists in the tree, or will be created immediately afterward.** To see why this is true, assume we are processing suffix α which is not a leaf (i.e., α either ends in the middle of an edge, or is a node with two or more children), yet the character(s) following α is/are not S[i+1]. If α ends at a node, the suffix link must already exist since T*i* is a correct suffix tree. If α ends in the middle of an edge, we must now split the edge and create a new node. Now our algorithm will try to add character S[i+1] to the next suffix, β such that α = c β. If β ends at a node, we simply add a suffix link from the newly created node to this node. If β ends in the middle of an edge, the character following β is not S[i+1], i.e., a new node will have to be created, and this node will be the target for the suffix link from the previously created node. Thus, suffix links will be correctly created. Furthermore, the process of following from suffix to suffix can use the same approach we used to find the longest common substring. Following the same logic, over the total execution of the algorithm, this process also only used O(n) computation.

Putting all these observation together we get an O(n) algorithm for constructing suffix trees.

Here is an example of how the algorithm works.

In this example, we build the suffix tree for string ABCABCD. At step I we simply build the tree for the first prefix, comprising a single suffix (A), and create a first leaf in the tree. By the observation above, this leaf will not need to be updated again, rather the string will be filled in automatically. In step II, we add another suffix (B), the suffix AB having been automatically added by the leaf rule, and create a new leaf. Same deal for step III. In step IV, like in the earlier steps, we implicitly add suffixes ABCA, BCA, and CA represented at the corresponding leaves. We just need to add suffix A (the 4th suffix). As in the previous iteration we were implicitly left at leaf 3, we progress to the root of the tree and try to find a path labeled A. This path already exists in the tree and we terminate this round. The current pointer is now placed at the end of this path (character A). Note that no new leaves or nodes have been created. At step V, we again know that suffixes ABCAB, BCAB, CAB are already in the tree at the corresponding leaves. We now try to add suffixes AB and B, starting from the last position we visited in the tree (character A highlighted in red in step IV). We find that suffix AB is already in the tree and terminate this round. Note that we did not need to check if suffix B is also found in the tree as we already know this fact to be true. The pointer in the tree has now progressed to the end of suffix 4, the B character highlighted in red. Step VI is the same as step V, progressing another position down the same edge. Finally, we reach step VII where we try to add the final character to all the suffixes not ending in a leaf. We start from the position where we last



ended (the red C character in panel VI), the end of suffix 4 (we started with this suffix in steps V, VI, and VII). As we do not find suffix ABCD in the tree we create a new node and a leaf for suffix 4, then progress to search for suffix 5. We do so by using the skip-count trick we developed earlier in the context of search. Specifically we find the parent of the node (the root in this case), and progress (blindly) down the path from the root that starts with B for two characters (length of ABC minus the first character). At this position we again try to add character D to the tree, and since it is not found, we create a new node and leaf 5. The newly created node becomes the target of the suffix link from the previously created node. We then follow a similar approach (move to the root, follow path labeled C one character down) to insert the 6ths suffix (CD), and add a suffix link for the previously created node. Finally, we insert the 7th suffix (D) in the tree, and create a suffix link from the previous node to the root..

Throughout this approach we see that at each stage we either created a leaf, or terminated the execution of the algorithm as soon as we found that a suffix existed in the tree.

Note that when creating the suffix links, the node to which the link will point to may already exist in the tree (see below), however we are still guaranteed that we will visit that node immediately after creating the new node. The example below follows the insertion of character D in the suffix tree constructed for the string ABCACDABC. In the last panel, the red node already existed in the tree when we try to fill in the suffix link for the newly created node (red node in the second to last panel).

S=ABCACDABCD